LanguageManual UDF

Hive Operators and User-Defined Functions (UDFs)

Case-insensitive
All Hive keywords are case-insensitive, including the names of Hive operators and functions.

In Beeline or the CLI, use the commands below to show the latest documentation:

```
SHOW FUNCTIONS;
DESCRIBE FUNCTION <function_name>;
DESCRIBE FUNCTION EXTENDED <function_name>;
```

Bug for expression caching when UDF nested in UDF or function
When hive.cache.expr.evaluation is set to true (which is the default) a UDF can give incorrect results if it is nested in another UDF or a Hive function. This bug affects releases 0.12.0, 0.13.0, and 0.13.1. Release 0.14.0 fixed the bug (HIVE-7314).

The problem relates to the UDF's implementation of the getDisplayString method, as discussed in the Hive user mailing list.

Built-in Operators

Operators Precedences

Example	Operators	Description
A[B] , A.identifier	bracket_op([]), dot(.)	element selector, dot
-A	unary(+), unary(-), unary(~)	unary prefix operators
A IS [NOT] (NULL|TRUE|FALSE)	IS NULL,IS NOT NULL, ...	unary suffix
A ^ B	bitwise xor(^)	bitwise xor
A * B	star(*), divide(/), mod(%), div(DIV)	multiplicative operators
A + B	plus(+), minus(-)	additive operators
A || B	string concatenate(||)	string concatenate
A & B	bitwise and(&)	bitwise and
A | B	bitwise or(|)	bitwise or

Relational Operators

The following operators compare the passed operands and generate a TRUE or FALSE value depending on whether the comparison between the operands holds.

Operator	Operand types	Description
A = B	All primitive types	TRUE if expression A is equal to expression B otherwise FALSE.
A == B	All primitive types	Synonym for the = operator.
A <=> B	All primitive types	Returns same result with EQUAL(=) operator for non-null operands, but returns TRUE if both are NULL, FALSE if one of the them is NULL. (As of version 0.9.0.)
A <> B	All primitive types	NULL if A or B is NULL, TRUE if expression A is NOT equal to expression B, otherwise FALSE.
A != B	All primitive types	Synonym for the <> operator.
A < B	All primitive types	NULL if A or B is NULL, TRUE if expression A is less than expression B, otherwise FALSE.
A <= B	All primitive types	NULL if A or B is NULL, TRUE if expression A is less than or equal to expression B, otherwise FALSE.
A > B	All primitive types	NULL if A or B is NULL, TRUE if expression A is greater than expression B, otherwise FALSE.
A >= B	All primitive types	NULL if A or B is NULL, TRUE if expression A is greater than or equal to expression B, otherwise FALSE.
A [NOT] BETWEEN B AND C	All primitive types	NULL if A, B or C is NULL, TRUE if A is greater than or equal to B AND A less than or equal to C, otherwise FALSE. This can be inverted by using the NOT keyword. (As of version 0.9.0.)
A IS NULL	All types	TRUE if expression A evaluates to NULL, otherwise FALSE.

A IS NOT NULL	All types	FALSE if expression A evaluates to NULL, otherwise TRUE.
A IS [NOT] (TRUE|FALSE)	Boolean types	Evaluates to TRUE only if A mets the condition. (since:3.0.0) Note: NULL is UNKNOWN, and because of that (UNKNOWN IS TRUE) and (UNKNOWN IS FALSE) both evaluates to FALSE.
A [NOT] LIKE B	strings	NULL if A or B is NULL, TRUE if string A matches the SQL simple regular expression B, otherwise FALSE. The comparison is done character by character. The _ character in B matches any character in A (similar to . in posix regular expressions) while the % character in B matches an arbitrary number of characters in A (similar to .* in posix regular expressions). For example, 'foobar' like 'foo' evaluates to FALSE whereas 'foobar' like 'foo_ _ _' evaluates to TRUE and so does 'foobar' like 'foo%'.
A RLIKE B	strings	NULL if A or B is NULL, TRUE if any (possibly empty) substring of A matches the Java regular expression B, otherwise FALSE. For example, 'foobar' RLIKE 'foo' evaluates to TRUE and so does 'foobar' RLIKE '^f.*r$'.
A REGEXP B	strings	Same as RLIKE.

Arithmetic Operators

The following operators support various common arithmetic operations on the operands. All return number types; if any of the operands are NULL, then the result is also NULL.

Operator	Operand types	Description
A + B	All number types	Gives the result of adding A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. For example since every integer is a float, therefore float is a containing type of integer so the + operator on a float and an int will result in a float.
A - B	All number types	Gives the result of subtracting B from A. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A * B	All number types	Gives the result of multiplying A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. Note that if the multiplication causing overflow, you will have to cast one of the operators to a type higher in the type hierarchy.
A / B	All number types	Gives the result of dividing A by B. The result is a double type in most cases. When A and B are both integers, the result is a double type except when the hive.compat configuration parameter is set to "0.13" or "latest" in which case the result is a decimal type.
A DIV B	Integer types	Gives the integer part resulting from dividing A by B. E.g 17 div 3 results in 5.
A % B	All number types	Gives the reminder resulting from dividing A by B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A & B	All number types	Gives the result of bitwise AND of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A | B	All number types	Gives the result of bitwise OR of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A ^ B	All number types	Gives the result of bitwise XOR of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
~A	All number types	Gives the result of bitwise NOT of A. The type of the result is the same as the type of A.

Logical Operators

The following operators provide support for creating logical expressions. All of them return boolean TRUE, FALSE, or NULL depending upon the boolean values of the operands. NULL behaves as an "unknown" flag, so if the result depends on the state of an unknown, the result itself is

unknown.

Operator	Operand types	Description
A AND B	boolean	TRUE if both A and B are TRUE, otherwise FALSE. NULL if A or B is NULL.
A OR B	boolean	TRUE if either A or B or both are TRUE, FALSE OR NULL is NULL, otherwise FALSE.
NOT A	boolean	TRUE if A is FALSE or NULL if A is NULL. Otherwise FALSE.
! A	boolean	Same as NOT A.
A IN (val1, val2, ...)	boolean	TRUE if A is equal to any of the values. As of Hive 0.13 subqueries are supported in IN statements.
A NOT IN (val1, val2, ...)	boolean	TRUE if A is not equal to any of the values. As of Hive 0.13 subqueries are supported in NOT IN statements.
[NOT] EXISTS (subquery)		TRUE if the the subquery returns at least one row. Supported as of Hive 0.13.

String Operators

Operator	Operand types	Description
A || B	strings	Concatenates the operands - shorthand for `concat(A,B)` . Supported as of Hive 2.2.0.

Complex Type Constructors

The following functions construct instances of complex types.

Constructor Function	Operands	Description
map	(key1, value1, key2, value2, ...)	Creates a map with the given key/value pairs.
struct	(val1, val2, val3, ...)	Creates a struct with the given field values. Struct field names will be col1, col2, ...
named_struct	(name1, val1, name2, val2, ...)	Creates a struct with the given field names and values. (As of Hive 0.8.0.)
array	(val1, val2, ...)	Creates an array with the given elements.
create_union	(tag, val1, val2, ...)	Creates a union type with the value that is being pointed to by the tag parameter.

Operators on Complex Types

The following operators provide mechanisms to access elements in Complex Types.

Operator	Operand types	Description
A[n]	A is an Array and n is an int	Returns the nth element in the array A. The first element has index 0. For example, if A is an array comprising of ['foo', 'bar'] then A[0] returns 'foo' and A[1] returns 'bar'.
M[key]	M is a Map<K, V> and key has type K	Returns the value corresponding to the key in the map. For example, if M is a map comprising of {'f' -> 'foo', 'b' -> 'bar', 'all' -> 'foobar'} then M['all'] returns 'foobar'.
S.x	S is a struct	Returns the x field of S. For example for the struct foobar {int foo, int bar}, foobar.foo returns the integer stored in the foo field of the struct.

Built-in Functions

Mathematical Functions

The following built-in mathematical functions are supported in Hive; most return NULL when the argument(s) are NULL:

Return Type	Name (Signature)	Description
DOUBLE	round(DOUBLE a)	Returns the rounded `BIGINT` value of a.
DOUBLE	round(DOUBLE a, INT d)	Returns a rounded to d decimal places.
DOUBLE	bround(DOUBLE a)	Returns the rounded BIGINT value of a using HALF_EVEN rounding mode (as of Hive 1.3.0, 2.0.0). Also known as Gaussian rounding or bankers' rounding. Example: bround(2.5) = 2, bround(3.5) = 4.
DOUBLE	bround(DOUBLE a, INT d)	Returns a rounded to d decimal places using HALF_EVEN rounding mode (as of Hive 1.3.0, 2.0.0). Example: bround(8.25, 1) = 8.2, bround(8.35, 1) = 8.4.
BIGINT	floor(DOUBLE a)	Returns the maximum `BIGINT` value that is equal to or less than a.
BIGINT	ceil(DOUBLE a), ceiling(DOUBLE a)	Returns the minimum BIGINT value that is equal to or greater than a.
DOUBLE	rand(), rand(INT seed)	Returns a random number (that changes from row to row) that is distributed uniformly from 0 to 1. Specifying the seed will make sure the generated random number sequence is deterministic.
DOUBLE	exp(DOUBLE a), exp(DECIMAL a)	Returns e^a where e is the base of the natural logarithm. Decimal version added in Hive 0.13.0.
DOUBLE	ln(DOUBLE a), ln(DECIMAL a)	Returns the natural logarithm of the argument a. Decimal version added in Hive 0.13.0.
DOUBLE	log10(DOUBLE a), log10(DECIMAL a)	Returns the base-10 logarithm of the argument a. Decimal version added in Hive 0.13.0.
DOUBLE	log2(DOUBLE a), log2(DECIMAL a)	Returns the base-2 logarithm of the argument a. Decimal version added in Hive 0.13.0.
DOUBLE	log(DOUBLE base, DOUBLE a) log(DECIMAL base, DECIMAL a)	Returns the base-`base` logarithm of the argument a. Decimal versions added in Hive 0.13.0.
DOUBLE	pow(DOUBLE a, DOUBLE p), power(DOUBLE a, DOUBLE p)	Returns a^p.
DOUBLE	sqrt(DOUBLE a), sqrt(DECIMAL a)	Returns the square root of a. Decimal version added in Hive 0.13.0.
STRING	bin(BIGINT a)	Returns the number in binary format (see http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_bin).
STRING	hex(BIGINT a) hex(STRING a) hex(BINARY a)	If the argument is an `INT` or `binary`, `hex` returns the number as a `STRING` in hexadecimal format. Otherwise if the number is a `STRING`, it converts each character into its hexadecimal representation and returns the resulting `STRING`. (See http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_hex, `BINARY` version as of Hive 0.12.0.)
BINARY	unhex(STRING a)	Inverse of hex. Interprets each pair of characters as a hexadecimal number and converts to the byte representation of the number. (`BINARY` version as of Hive 0.12.0, used to return a string.)
STRING	conv(BIGINT num, INT from_base, INT to_base), conv(STRING num, INT from_base, INT to_base)	Converts a number from a given base to another (see http://dev.mysql.com/doc/refman/5.0/en/mathematical-functions.html#function_conv).
DOUBLE	abs(DOUBLE a)	Returns the absolute value.
INT or DOUBLE	pmod(INT a, INT b), pmod(DOUBLE a, DOUBLE b)	Returns the positive value of `a mod b`.
DOUBLE	sin(DOUBLE a), sin(DECIMAL a)	Returns the sine of a (a is in radians). Decimal version added in Hive 0.13.0.
DOUBLE	asin(DOUBLE a), asin(DECIMAL a)	Returns the arc sin of a if -1<=a<=1 or NULL otherwise. Decimal version added in Hive 0.13.0.
DOUBLE	cos(DOUBLE a), cos(DECIMAL a)	Returns the cosine of a (a is in radians). Decimal version added in Hive 0.13.0.

DOUBLE	acos(DOUBLE a), acos(DECIMAL a)	Returns the arccosine of a if -1<=a<=1 or NULL otherwise. Decimal version added in Hive 0.13.0.
DOUBLE	tan(DOUBLE a), tan(DECIMAL a)	Returns the tangent of a (a is in radians). Decimal version added in Hive 0.13.0.
DOUBLE	atan(DOUBLE a), atan(DECIMAL a)	Returns the arctangent of a. Decimal version added in Hive 0.13.0.
DOUBLE	degrees(DOUBLE a), degrees(DECIMAL a)	Converts value of a from radians to degrees. Decimal version added in Hive 0.13.0.
DOUBLE	radians(DOUBLE a), radians(DOUBLE a)	Converts value of a from degrees to radians. Decimal version added in Hive 0.13.0.
INT or DOUBLE	positive(INT a), positive(DOUBLE a)	Returns a.
INT or DOUBLE	negative(INT a), negative(DOUBLE a)	Returns -a.
DOUBLE or INT	sign(DOUBLE a), sign(DECIMAL a)	Returns the sign of a as '1.0' (if a is positive) or '-1.0' (if a is negative), '0.0' otherwise. The decimal version returns INT instead of DOUBLE. Decimal version added in Hive 0.13.0.
DOUBLE	e()	Returns the value of e.
DOUBLE	pi()	Returns the value of pi.
BIGINT	factorial(INT a)	Returns the factorial of a (as of Hive 1.2.0). Valid a is [0..20].
DOUBLE	cbrt(DOUBLE a)	Returns the cube root of a double value (as of Hive 1.2.0).
INT BIGINT	shiftleft(TINYINT|SMALLINT|INT a, INT b) shiftleft(BIGINT a, INT b)	Bitwise left shift (as of Hive 1.2.0). Shifts a b positions to the left. Returns int for tinyint, smallint and int a. Returns bigint for bigint a.
INT BIGINT	shiftright(TINYINT|SMALLINT|INT a, INT b) shiftright(BIGINT a, INT b)	Bitwise right shift (as of Hive 1.2.0). Shifts a b positions to the right. Returns int for tinyint, smallint and int a. Returns bigint for bigint a.
INT BIGINT	shiftrightunsigned(TINYINT|SMALLINT|INT a, INT b), shiftrightunsigned(BIGINT a, INT b)	Bitwise unsigned right shift (as of Hive 1.2.0). Shifts a b positions to the right. Returns int for tinyint, smallint and int a. Returns bigint for bigint a.
T	greatest(T v1, T v2, ...)	Returns the greatest value of the list of values (as of Hive 1.1.0). Fixed to return NULL when one or more arguments are NULL, and strict type restriction relaxed, consistent with ">" operator (as of Hive 2.0.0).
T	least(T v1, T v2, ...)	Returns the least value of the list of values (as of Hive 1.1.0). Fixed to return NULL when one or more arguments are NULL, and strict type restriction relaxed, consistent with "<" operator (as of Hive 2.0.0).
INT	width_bucket(NUMERIC expr, NUMERIC min_value, NUMERIC max_value, INT num_buckets)	Returns an integer between 0 and num_buckets+1 by mapping expr into the ith equally sized bucket. Buckets are made by dividing [min_value, max_value] into equally sized regions. If expr < min_value, return 1, if expr > max_value return num_buckets+1. See https://docs.oracle.com/cd/B19306_01/server.102/b14200/functions214.htm (as of Hive 3.0.0)

Mathematical Functions and Operators for Decimal Datatypes

> **Version**
> The decimal datatype was introduced in Hive 0.11.0 (HIVE-2693).

All regular arithmetic operators (such as +, -, *, /) and relevant mathematical UDFs (Floor, Ceil, Round, and many more) have been updated to handle decimal types. For a list of supported UDFs, see Mathematical UDFs in Hive Data Types.

Collection Functions

The following built-in collection functions are supported in Hive:

Return Type	Name(Signature)	Description
int	size(Map<K.V>)	Returns the number of elements in the map type.
int	size(Array<T>)	Returns the number of elements in the array type.
array<K>	map_keys(Map<K.V>)	Returns an unordered array containing the keys of the input map.
array<V>	map_values(Map<K.V>)	Returns an unordered array containing the values of the input map.
boolean	array_contains(Array<T>, value)	Returns TRUE if the array contains value.
array<t>	sort_array(Array<T>)	Sorts the input array in ascending order according to the natural ordering of the array elements and returns it (as of version 0.9.0).

Type Conversion Functions

The following type conversion functions are supported in Hive:

Return Type	Name(Signature)	Description
binary	binary(string|binary)	Casts the parameter into a binary.
Expected "=" to follow "type"	cast(expr as <type>)	Converts the results of the expression expr to <type>. For example, cast('1' as BIGINT) will convert the string '1' to its integral representation. A null is returned if the conversion does not succeed. If cast(expr as boolean) Hive returns true for a non-empty string.

Date Functions

The following built-in date functions are supported in Hive:

Return Type	Name(Signature)	Description
string	from_unixtime(bigint unixtime[, string format])	Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the format of "1970-01-01 00:00:00".
bigint	unix_timestamp()	Gets current Unix timestamp in seconds. This function is not deterministic and its value is not fixed for the scope of a query execution, therefore prevents proper optimization of queries - this has been deprecated since 2.0 in favour of CURRENT_TIMESTAMP constant.
bigint	unix_timestamp(string date)	Converts time string in format `yyyy-MM-dd HH:mm:ss` to Unix timestamp (in seconds), using the default timezone and the default locale, return 0 if fail: unix_timestamp('2009-03-20 11:30:01') = 1237573801
bigint	unix_timestamp(string date, string pattern)	Convert time string with given pattern (see [http://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html]) to Unix time stamp (in seconds), return 0 if fail: unix_timestamp('2009-03-20', 'yyyy-MM-dd') = 1237532400.
pre 2.1.0: string *2.1.0 on:* date	to_date(string timestamp)	Returns the date part of a timestamp string (pre-Hive 2.1.0): to_date("1970-01-01 00:00:00") = "1970-01-01". As of Hive 2.1.0, returns a date object. Prior to Hive 2.1.0 (HIVE-13248) the return type was a String because no Date type existed when the method was created.
int	year(string date)	Returns the year part of a date or a timestamp string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970.
int	quarter(date/timestamp/string)	Returns the quarter of the year for a date, timestamp, or string in the range 1 to 4 (as of Hive 1.3.0). Example: quarter('2015-04-08') = 2.

int	month(string date)	Returns the month part of a date or a timestamp string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11.
int	day(string date) dayofmonth(date)	Returns the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1.
int	hour(string date)	Returns the hour of the timestamp: hour('2009-07-30 12:58:59') = 12, hour('12:58:59') = 12.
int	minute(string date)	Returns the minute of the timestamp.
int	second(string date)	Returns the second of the timestamp.
int	weekofyear(string date)	Returns the week number of a timestamp string: weekofyear("1970-11-01 00:00:00") = 44, weekofyear("1970-11-01") = 44.
int	extract(field FROM source)	Retrieve fields such as days or hours from source (as of Hive 2.2.0). Source must be a date, timestamp, interval or a string that can be converted into either a date or timestamp. Supported fields include: day, dayofweek, hour, minute, month, quarter, second, week and year. Examples: 1. select extract(month from "2016-10-20") results in 10. 2. select extract(hour from "2016-10-20 05:06:07") results in 5. 3. select extract(dayofweek from "2016-10-20 05:06:07") results in 5. 4. select extract(month from interval '1-3' year to month) results in 3. 5. select extract(minute from interval '3 12:20:30' day to second) results in 20.
int	datediff(string enddate, string startdate)	Returns the number of days from startdate to enddate: datediff('2009-03-01', '2009-02-27') = 2.
pre 2.1.0: string *2.1.0 on:* date	date_add(date/timestamp/string startdate, tinyint/smallint/int days)	Adds a number of days to startdate: date_add('2008-12-31', 1) = '2009-01-01'. Prior to Hive 2.1.0 (HIVE-13248) the return type was a String because no Date type existed when the method was created.
pre 2.1.0: string *2.1.0 on:* date	date_sub(date/timestamp/string startdate, tinyint/smallint/int days)	Subtracts a number of days to startdate: date_sub('2008-12-31', 1) = '2008-12-30'. Prior to Hive 2.1.0 (HIVE-13248) the return type was a String because no Date type existed when the method was created.
timestamp	from_utc_timestamp({any primitive type}*, string timezone)	Coverts a timestamp* in UTC to a given timezone (as of Hive 0.8.0). * timestamp is a primitive type, including timestamp/date, tinyint/smallint/int/bigint, float/double and decimal. Fractional values are considered as seconds. Integer values are considered as milliseconds.. E.g from_utc_timestamp(2592000.0,'PST'), from_utc_timestamp(2592000000,'PST') and from_utc_timestamp(timestamp '1970-01-30 16:00:00','PST') all return the timestamp 1970-01-30 08:00:00
timestamp	to_utc_timestamp({any primitive type} ts, string timezone)	Coverts a timestamp* in a given timezone to UTC (as of Hive 0.8.0). * timestamp is a primitive type, including timestamp/date, tinyint/smallint/int/bigint, float/double and decimal. Fractional values are considered as seconds. Integer values are considered as milliseconds.. E.g to_utc_timestamp(2592000.0,'PST'), to_utc_timestamp(2592000000,'PST') and to_utc_timestamp(timestamp '1970-01-30 16:00:00','PST') all return the timestamp 1970-01-31 00:00:00
date	current_date	Returns the current date at the start of query evaluation (as of Hive 1.2.0). All calls of current_date within the same query return the same value.
timestamp	current_timestamp	Returns the current timestamp at the start of query evaluation (as of Hive 1.2.0). All calls of current_timestamp within the same query return the same value.
string	add_months(string start_date, int num_months)	Returns the date that is num_months after start_date (as of Hive 1.1.0). start_date is a string, date or timestamp. num_months is an integer. The time part of start_date is ignored. If start_date is the last day of the month or if the resulting month has fewer days than the day component of start_date, then the result is the last day of the resulting month. Otherwise, the result has the same day component as start_date.
string	last_day(string date)	Returns the last day of the month which the date belongs to (as of Hive 1.1.0). date is a string in the format 'yyyy-MM-dd HH:mm:ss' or 'yyyy-MM-dd'. The time part of date is ignored.

string	next_day(string start_date, string day_of_week)	Returns the first date which is later than start_date and named as day_of_week (as of Hive 1.2.0). start_date is a string/date/timestamp. day_of_week is 2 letters, 3 letters or full name of the day of the week (e.g. Mo, tue, FRIDAY). The time part of start_date is ignored. Example: next_day('2015-01-14', 'TU') = 2015-01-20.
string	trunc(string date, string format)	Returns date truncated to the unit specified by the format (as of Hive 1.2.0). Supported formats: MONTH/MON/MM, YEAR/YYYY/YY. Example: trunc('2015-03-17', 'MM') = 2015-03-01.
double	months_between(date1, date2)	Returns number of months between dates date1 and date2 (as of Hive 1.2.0). If date1 is later than date2, then the result is positive. If date1 is earlier than date2, then the result is negative. If date1 and date2 are either the same days of the month or both last days of months, then the result is always an integer. Otherwise the UDF calculates the fractional portion of the result based on a 31-day month and considers the difference in time components date1 and date2. date1 and date2 type can be date, timestamp or string in the format 'yyyy-MM-dd' or 'yyyy-MM-dd HH:mm:ss'. The result is rounded to 8 decimal places. Example: months_between('1997-02-28 10:30:00', '1996-10-30') = 3.94959677
string	date_format(date/timestamp/string ts, string fmt)	Converts a date/timestamp/string to a value of string in the format specified by the date format fmt (as of Hive 1.2.0). Supported formats are Java SimpleDateFormat formats – https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html. The second argument fmt should be constant. Example: date_format('2015-04-08', 'y') = '2015'. date_format can be used to implement other UDFs, e.g.: • dayname(date) is date_format(date, 'EEEE') • dayofyear(date) is date_format(date, 'D')

Conditional Functions

Return Type	Name(Signature)	Description
T	if(boolean testCondition, T valueTrue, T valueFalseOrNull)	Returns valueTrue when testCondition is true, returns valueFalseOrNull otherwise.
boolean	isnull(a)	Returns true if a is NULL and false otherwise.
boolean	isnotnull (a)	Returns true if a is not NULL and false otherwise.
T	nvl(T value, T default_value)	Returns default value if value is null else returns value (as of HIve 0.11).
T	COALESCE(T v1, T v2, ...)	Returns the first v that is not NULL, or NULL if all v's are NULL.
T	CASE a WHEN b THEN c [WHEN d THEN e]* [ELSE f] END	When a = b, returns c; when a = d, returns e; else returns f.
T	CASE WHEN a THEN b [WHEN c THEN d]* [ELSE e] END	When a = true, returns b; when c = true, returns d; else returns e.
T	nullif(a, b)	Returns NULL if a=b; otherwise returns a (as of Hive 2.2.0). Shorthand for: CASE WHEN a = b then NULL else a
void	assert_true(boolean condition)	Throw an exception if 'condition' is not true, otherwise return null (as of Hive 0.8.0). For example, select assert_true (2<1).

String Functions

The following built-in String functions are supported in Hive:

Return Type	Name(Signature)	Description
int	ascii(string str)	Returns the numeric value of the first character of str.
string	base64(binary bin)	Converts the argument from binary to a base 64 string (as of Hive 0.12.0
int	character_length(string str)	Returns the number of UTF-8 characters contained in str (as of Hive 2.2. The function char_length is shorthand for this function.

string	chr(bigint|double A)	Returns the ASCII character having the binary equivalent to A (as of Hive .0 and 2.1.0). If A is larger than 256 the result is equivalent to chr(A % 25 Example: select chr(88); returns "X".
string	concat(string|binary A, string|binary B...)	Returns the string or bytes resulting from concatenating the strings or by passed in as parameters in order. For example, concat('foo', 'bar') results 'foobar'. Note that this function can take any number of input strings.
array<struct<string,double>>	context_ngrams(array<array<string>>, array<string>, int K, int pf)	Returns the top-k contextual N-grams from a set of tokenized sentences, given a string of "context". See StatisticsAndDataMining for more information.
string	concat_ws(string SEP, string A, string B...)	Like concat() above, but with custom separator SEP.
string	concat_ws(string SEP, array<string>)	Like concat_ws() above, but taking an array of strings. (as of Hive 0.9.0)
string	decode(binary bin, string charset)	Decodes the first argument into a String using the provided character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'). If either argument is null, the result will also be null. (As of Hive 12.0.)
string	elt(N int,str1 string,str2 string,str3 string,...)	Return string at index number. For example elt(2,'hello','world') returns 'world'. Returns NULL if N is less than 1 or greater than the number of arguments. (see https://dev.mysql.com/doc/refman/5.7/en/string-functions.html#funct _elt)
binary	encode(string src, string charset)	Encodes the first argument into a BINARY using the provided character s (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'). If either argument is null, the result will also be null. (As of Hive 12.0.)
int	field(val T,val1 T,val2 T,val3 T,...)	Returns the index of val in the val1,val2,val3,... list or 0 if not found. For example field('world','say','hello','world') returns 3. All primitive types are supported, arguments are compared using str.equals(x). If val is NULL, the return value is 0. (see https://dev.mysql.com/doc/refman/5.7/en/string-functions.html#funct _field)
int	find_in_set(string str, string strList)	Returns the first occurance of str in strList where strList is a comma-delimited string. Returns null if either argument is null. Returns 0 the first argument contains any commas. For example, find_in_set('ab', 'abc,b,ab,c,def') returns 3.
string	format_number(number x, int d)	Formats the number X to a format like '#,###,###.##', rounded to D decir places, and returns the result as a string. If D is 0, the result has no decir point or fractional part. (As of Hive 0.10.0; bug with float types fixed in Hi 0.14.0, decimal type support added in Hive 0.14.0)
string	get_json_object(string json_string, string path)	Extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It will return null if the inpu json string is invalid. **NOTE: The json path can only have the characte [0-9a-z_], i.e., no upper-case or special characters. Also, the keys *cannot start with numbers.*** This is due to restrictions on Hive column names.
boolean	in_file(string str, string filename)	Returns true if the string str appears as an entire line in filename.
int	instr(string str, string substr)	Returns the position of the first occurrence of `substr` in `str`. Returns n if either of the arguments are `null` and returns `0` if `substr` could not be found in `str`. Be aware that this is not zero based. The first character in has index 1.
int	length(string A)	Returns the length of the string.
int	locate(string substr, string str[, int pos])	Returns the position of the first occurrence of substr in str after position p
string	lower(string A) lcase(string A)	Returns the string resulting from converting all characters of B to lower c For example, lower('fOoBaR') results in 'foobar'.

string	lpad(string str, int len, string pad)	Returns str, left-padded with pad to a length of len. If str is longer than le the return value is shortened to len characters. In case of empty pad strir the return value is null.
string	ltrim(string A)	Returns the string resulting from trimming spaces from the beginning(left hand side) of A. For example, ltrim(' foobar ') results in 'foobar '.
array<struct<string,double>>	ngrams(array<array<string>>, int N, int K, int pf)	Returns the top-k N-grams from a set of tokenized sentences, such as th returned by the sentences() UDAF. See StatisticsAndDataMining for mor information.
int	octet_length(string str)	Returns the number of octets required to hold the string str in UTF-8 encoding (since Hive 2.2.0). Note that octet_length(str) can be larger tha character_length(str).
string	parse_url(string urlString, string partToExtract [, string keyToExtract])	Returns the specified part from the URL. Valid values for partToExtract include HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, a USERINFO. For example, parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'HOST returns 'facebook.com'. Also a value of a particular key in QUERY can be extracted by providing the key as the third argument, for example, parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUER 'k1') returns 'v1'.
string	printf(String format, Obj... args)	Returns the input formatted according do printf-style format strings (as of Hive 0.9.0).
string	regexp_extract(string subject, string pattern, int index)	Returns the string extracted using the pattern. For example, regexp_extract('foothebar', 'foo(.*?)(bar)', 2) returns 'bar.' Note that some care is necessary in using predefined character classes: using '\s' as the second argument will match the letter s; '\\s' is necessary to match whitespace, etc. The 'index' parameter is the Java regex Matcher group(method index. See docs/api/java/util/regex/Matcher.html for more information on the 'index' or Java regex group() method.
string	regexp_replace(string INITIAL_STRING, string PATTERN, string REPLACEMENT)	Returns the string resulting from replacing all substrings in INITIAL_STR that match the java regular expression syntax defined in PATTERN with instances of REPLACEMENT. For example, regexp_replace("foobar", "oo|ar", "") returns 'fb.' Note that some care is necessary in using predefi character classes: using '\s' as the second argument will match the letter '\\s' is necessary to match whitespace, etc.
string	repeat(string str, int n)	Repeats str n times.
string	replace(string A, string OLD, string NEW)	Returns the string A with all non-overlapping occurrences of OLD replace with NEW (as of Hive 1.3.0 and 2.1.0). Example: select replace("ababab "abab", "Z"); returns "Zab".
string	reverse(string A)	Returns the reversed string.
string	rpad(string str, int len, string pad)	Returns str, right-padded with pad to a length of len. If str is longer than l the return value is shortened to len characters. In case of empty pad strir the return value is null.
string	rtrim(string A)	Returns the string resulting from trimming spaces from the end(right hand side) of A. For example, rtrim(' foobar ') results in ' foobar'.
array<array<string>>	sentences(string str, string lang, string locale)	Tokenizes a string of natural language text into words and sentences, wh each sentence is broken at the appropriate sentence boundary and retur as an array of words. The 'lang' and 'locale' are optional arguments. For example, sentences('Hello there! How are you?') returns (("Hello", "there ("How", "are", "you")).
string	space(int n)	Returns a string of n spaces.
array	split(string str, string pat)	Splits str around pat (pat is a regular expression).
map<string,string>	str_to_map(text[, delimiter1, delimiter2])	Splits text into key-value pairs using two delimiters. Delimiter1 separates into K-V pairs, and Delimiter2 splits each K-V pair. Default delimiters are for delimiter1 and ':' for delimiter2.

string	substr(string|binary A, int start) substring(string|binary A, int start)	Returns the substring or slice of the byte array of A starting from start position till the end of string A. For example, substr('foobar', 4) results in (see [http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#funct substr]).
string	substr(string|binary A, int start, int len) substring(string|binary A, int start, int len)	Returns the substring or slice of the byte array of A starting from start position with length len. For example, substr('foobar', 4, 1) results in 'b' (s http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_su]).
string	substring_index(string A, string delim, int count)	Returns the substring from string A before count occurrences of the delin delim (as of Hive 1.3.0). If count is positive, everything to the left of the fi delimiter (counting from the left) is returned. If count is negative, everythi to the right of the final delimiter (counting from the right) is returned. Substring_index performs a case-sensitive match when searching for de Example: substring_index('www.apache.org', '.', 2) = 'www.apache'.
string	translate(string|char|varchar input, string|char|varchar from, string|char|varchar to)	Translates the input string by replacing the characters present in the `fro` ring with the corresponding characters in the `to` string. This is similar to t `translate` function in PostgreSQL. If any of the parameters to this UDF are NULL, the result is NULL as well. (Available as of Hive 0.10.0, for str types) Char/varchar support added as of Hive 0.14.0.
string	trim(string A)	Returns the string resulting from trimming spaces from both ends of A. F example, trim(' foobar ') results in 'foobar'
binary	unbase64(string str)	Converts the argument from a base 64 string to BINARY. (As of Hive 0.1)
string	upper(string A) ucase(string A)	Returns the string resulting from converting all characters of A to upper c For example, upper('fOoBaR') results in 'FOOBAR'.
string	initcap(string A)	Returns string, with the first letter of each word in uppercase, all other let in lowercase. Words are delimited by whitespace. (As of Hive 1.1.0.)
int	levenshtein(string A, string B)	Returns the Levenshtein distance between two strings (as of Hive 1.2.0). example, levenshtein('kitten', 'sitting') results in 3.
string	soundex(string A)	Returns soundex code of the string (as of Hive 1.2.0). For example, soundex('Miller') results in M460.

Data Masking Functions

The following built-in data masking functions are supported in Hive:

Return Type	Name(Signature)	Description
string	mask(string str[, string upper[, string lower[, string number]]])	Returns a masked version of str (as of Hive 2.1.0). By default, upper case letters are converted to "X", lower case letters are converted to "x" and numbers are converted to "n". For example mask("abcd-EFGH-8765-4321") results in xxxx-XXXX-nnnn-nnnn. You can override the characters used in the mask by supplying additional arguments: the second argument controls the mask character for upper case letters, the third argument for lower case letters and the fourth argument for numbers. For example, mask("abcd-EFGH-8765-4321", "U", "l", "#") results in llll-UUUU-####-####.
string	mask_first_n(string str[, int n])	Returns a masked version of str with the first n values masked (as of Hive 2.1.0). Upper case letters are converted to "X", lower case letters are converted to "x" and numbers are converted to "n". For example, mask_first_n("1234-5678-8765-4321", 4) results in nnnn-5678-8765-4321.
string	mask_last_n(string str[, int n])	Returns a masked version of str with the last n values masked (as of Hive 2.1.0). Upper case letters are converted to "X", lower case letters are converted to "x" and numbers are converted to "n". For example, mask_last_n("1234-5678-8765-4321", 4) results in 1234-5678-8765-nnnn.
string	mask_show_first_n(string str[, int n])	Returns a masked version of str, showing the first n characters unmasked (as of Hive 2.1.0). Upper case letters are converted to "X", lower case letters are converted to "x" and numbers are converted to "n". For example, mask_show_first_n("1234-5678-8765-4321", 4) results in 1234-nnnn-nnnn-nnnn.

string	mask_show_last_n(string str[, int n])	Returns a masked version of str, showing the last n characters unmasked (as of Hive 2.1.0). Upper case letters are converted to "X", lower case letters are converted to "x" and numbers are converted to "n". For example, mask_show_last_n("1234-5678-8765-4321", 4) results in nnnn-nnnn-nnnn-4321.
string	mask_hash(string|char|varchar str)	Returns a hashed value based on str (as of Hive 2.1.0). The hash is consistent and can be used to join masked values together across tables. This function returns null for non-string types.

Misc. Functions

Return Type	Name(Signature)	Description
varies	java_method(class, method[, arg1[, arg2..]])	Synonym for `reflect`. (As of Hive 0.9.0.)
varies	reflect(class, method[, arg1[, arg2..]])	Calls a Java method by matching the argument signature, using reflection. (As of Hive 0.7.0.) See Reflect (Generic) UDF for examples.
int	hash(a1[, a2...])	Returns a hash value of the arguments. (As of Hive 0.4.)
string	current_user()	Returns current user name from the configured authenticator manager (as of Hive 1.2.0). Could be the same as the user provided when connecting, but with some authentication managers (for example HadoopDefaultAuthenticator) it could be different.
string	logged_in_user()	Returns current user name from the session state (as of Hive 2.2.0). This is the username provided when connecting to Hive.
string	current_database()	Returns current database name (as of Hive 0.13.0).
string	md5(string/binary)	Calculates an MD5 128-bit checksum for the string or binary (as of Hive 1.3.0). The value is returned as a string of 32 hex digits, or NULL if the argument was NULL. Example: md5('ABC') = '902fbdd2b1df0c4f70b4a5d23525e932'.
string	sha1(string/binary) sha(string/binary)	Calculates the SHA-1 digest for string or binary and returns the value as a hex string (as of Hive 1.3.0). Example: sha1('ABC') = '3c01bdbb26f358bab27f267924aa2c9a03fcfdb8'.
bigint	crc32(string/binary)	Computes a cyclic redundancy check value for string or binary argument and returns bigint value (as of Hive 1.3.0). Example: crc32('ABC') = 2743272264.
string	sha2(string/binary, int)	Calculates the SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512) (as of Hive 1.3.0). The first argument is the string or binary to be hashed. The second argument indicates the desired bit length of the result, which must have a value of 224, 256, 384, 512, or 0 (which is equivalent to 256). SHA-224 is supported starting from Java 8. If either argument is NULL or the hash length is not one of the permitted values, the return value is NULL. Example: sha2('ABC', 256) = 'b5d4045c3f466fa91fe2cc6abe79232a1a57cdf104f7a26e716e0a1e2789df78'.
binary	aes_encrypt(input string/binary, key string/binary)	Encrypt input using AES (as of Hive 1.3.0). Key lengths of 128, 192 or 256 bits can be used. 192 and 256 bits keys can be used if Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files are installed. If either argument is NULL or the key length is not one of the permitted values, the return value is NULL. Example: base64(aes_encrypt('ABC', '1234567890123456')) = 'y6Ss+zCYObpCbgfWfyNWTw=='.
binary	aes_decrypt(input binary, key string/binary)	Decrypt input using AES (as of Hive 1.3.0). Key lengths of 128, 192 or 256 bits can be used. 192 and 256 bits keys can be used if Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files are installed. If either argument is NULL or the key length is not one of the permitted values, the return value is NULL. Example: aes_decrypt(unbase64('y6Ss+zCYObpCbgfWfyNWTw=='), '1234567890123456') = 'ABC'.
string	version()	Returns the Hive version (as of Hive 2.1.0). The string contains 2 fields, the first being a build number and the second being a build hash. Example: "select version();" might return "2.1.0.2.5.0.0-1245 r027527b9c5ce1a3d7d0b6d2e6de2378fb0c39232". Actual results will depend on your build.

xpath

The following functions are described in LanguageManual XPathUDF:

- xpath, xpath_short, xpath_int, xpath_long, xpath_float, xpath_double, xpath_number, xpath_string

get_json_object

A limited version of JSONPath is supported:

- $: Root object
- . : Child operator
- [] : Subscript operator for array
- * : Wildcard for []

Syntax not supported that's worth noticing:

- : Zero length string as key
- .. : Recursive descent
- @ : Current object/element
- () : Script expression
- ?() : Filter (script) expression.
- [,] : Union operator
- [start:end.step] : array slice operator

Example: src_json table is a single column (json), single row table:

```
+----+
                                json
+----+
{"store":
  {"fruit":\[{"weight":8,"type":"apple"},{"weight":9,"type":"pear"}],
   "bicycle":{"price":19.95,"color":"red"}
  },
 "email":"amy@only_for_json_udf_test.net",
 "owner":"amy"
}
+----+
```

The fields of the json object can be extracted using these queries:

```
hive> SELECT get_json_object(src_json.json, '$.owner') FROM src_json;
amy

hive> SELECT get_json_object(src_json.json, '$.store.fruit\[0]') FROM src_json;
{"weight":8,"type":"apple"}

hive> SELECT get_json_object(src_json.json, '$.non_exist_key') FROM src_json;
NULL
```

Built-in Aggregate Functions (UDAF)

The following built-in aggregate functions are supported in Hive:

Return Type	Name(Signature)	Description
BIGINT	count(*), count(expr), count(DISTINCT expr[, expr...])	count(*) - Returns the total number of retrieved rows, including rows containing NULL values. count(expr) - Returns the number of rows for which the supplied expression is non-NULL. count(DISTINCT expr[, expr]) - Returns the number of rows for which the supplied expression(s) are unique and non-NULL. Execution of this can be optimized with hive.optimize.distinct.rewrite.
DOUBLE	sum(col), sum(DISTINCT col)	Returns the sum of the elements in the group or the sum of the distinct values of the column in the group.

DOUBLE	avg(col), avg(DISTINCT col)	Returns the average of the elements in the group or the average of the distinct values of the column in the group.
DOUBLE	min(col)	Returns the minimum of the column in the group.
DOUBLE	max(col)	Returns the maximum value of the column in the group.
DOUBLE	variance(col), var_pop(col)	Returns the variance of a numeric column in the group.
DOUBLE	var_samp(col)	Returns the unbiased sample variance of a numeric column in the group.
DOUBLE	stddev_pop(col)	Returns the standard deviation of a numeric column in the group.
DOUBLE	stddev_samp(col)	Returns the unbiased sample standard deviation of a numeric column in the group.
DOUBLE	covar_pop(col1, col2)	Returns the population covariance of a pair of numeric columns in the group.
DOUBLE	covar_samp(col1, col2)	Returns the sample covariance of a pair of a numeric columns in the group.
DOUBLE	corr(col1, col2)	Returns the Pearson coefficient of correlation of a pair of a numeric columns in the group.
DOUBLE	percentile(BIGINT col, p)	Returns the exact p^{th} percentile of a column in the group (does not work with floating point types). p must be between 0 and 1. NOTE: A true percentile can only be computed for integer values. Use PERCENTILE_APPROX if your input is non-integral.
array<double>	percentile(BIGINT col, array(p_1 [, p_2]...))	Returns the exact percentiles p_1, p_2, ... of a column in the group (does not work with floating point types). p_i must be between 0 and 1. NOTE: A true percentile can only be computed for integer values. Use PERCENTILE_APPROX if your input is non-integral.
DOUBLE	percentile_approx(DOUBLE col, p [, B])	Returns an approximate p^{th} percentile of a numeric column (including floating point types) in the group. The B parameter controls approximation accuracy at the cost of memory. Higher values yield better approximations, and the default is 10,000. When the number of distinct values in col is smaller than B, this gives an exact percentile value.
array<double>	percentile_approx(DOUBLE col, array(p_1 [, p_2]...) [, B])	Same as above, but accepts and returns an array of percentile values instead of a single one.
double	regr_avgx(independent, dependent)	Equivalent to avg(dependent). As of Hive 2.2.0.
double	regr_avgy(independent, dependent)	Equivalent to avg(independent). As of Hive 2.2.0.
double	regr_count(independent, dependent)	Returns the number of non-null pairs used to fit the linear regression line. As of Hive 2.2.0.
double	regr_intercept(independent, dependent)	Returns the y-intercept of the linear regression line, i.e. the value of b in the equation dependent = a * independent + b. As of Hive 2.2.0.
double	regr_r2(independent, dependent)	Returns the coefficient of determination for the regression. As of Hive 2.2.0.
double	regr_slope(independent, dependent)	Returns the slope of the linear regression line, i.e. the value of a in the equation dependent = a * independent + b. As of Hive 2.2.0.
double	regr_sxx(independent, dependent)	Equivalent to regr_count(independent, dependent) * var_pop(dependent). As of Hive 2.2.0.
double	regr_sxy(independent, dependent)	Equivalent to regr_count(independent, dependent) * covar_pop(independent, dependent). As of Hive 2.2.0.
double	regr_syy(independent, dependent)	Equivalent to regr_count(independent, dependent) * var_pop(independent). As of Hive 2.2.0.
array<struct {'x','y'}>	histogram_numeric(col, b)	Computes a histogram of a numeric column in the group using b non-uniformly spaced bins. The output is an array of size b of double-valued (x,y) coordinates that represent the bin centers and heights
array	collect_set(col)	Returns a set of objects with duplicate elements eliminated.
array	collect_list(col)	Returns a list of objects with duplicates. (As of Hive 0.13.0.)

INTEGER	ntile(INTEGER x)	Divides an ordered partition into x groups called buckets and assigns a bucket number to each row in the partition. This allows easy calculation of tertiles, quartiles, deciles, percentiles and other common summary statistics. (As of Hive 0.11.0.)

Built-in Table-Generating Functions (UDTF)

Normal user-defined functions, such as concat(), take in a single input row and output a single output row. In contrast, table-generating functions transform a single input row to multiple output rows.

Row-set columns types	Name(Signature)	Description
T	explode(ARRAY<T> a)	Explodes an array to multiple rows. Returns a row-set with a single column (*col*), one row for each element from the array.
T_{key},T_{value}	explode(MAP<T_{key},T_{value}> m)	Explodes a map to multiple rows. Returns a row-set with a two columns (*key,value)* , one row for each key-value pair from the input map. (As of Hive 0.8.0.).
int,T	posexplode(ARRAY<T> a)	Explodes an array to multiple rows with additional positional column of *int* type (position of items in the original array, starting with 0). Returns a row-set with two columns (*pos,val*), one row for each element from the array.
T_1,...,T_n	inline(ARRAY<STRUCT<f_1:T_1,...,f_n:T_n>> a)	Explodes an array of structs to multiple rows. Returns a row-set with N columns (N = number of top level elements in the struct), one row per struct from the array. (As of Hive 0.10.)
T_1,...,$T_{n/r}$	stack(int r,T_1 V_1,...,$T_{n/r}$ V_n)	Breaks up *n* values V_1,...,V_n into *r* rows. Each row will have *n/r* columns. *r* must be constant.
$string_1$,..., $string_n$	json_tuple(string jsonStr,string k_1,...,string k_n)	Takes JSON string and a set of *n* keys, and returns a tuple of *n* values. This is a more efficient version of the `get_json_object` UDF because it can get multiple keys with just one call.
string $_1$,... ,$string_n$	parse_url_tuple(string urlStr,string p_1,...,string p_n)	Takes URL string and a set of *n* URL parts, and returns a tuple of *n* values. This is similar to the `parse_url()` UDF but can extract multiple parts at once out of a URL. Valid part names are: HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, USERINFO, QUERY:<KEY>.

Usage Examples

explode (array)

```
select explode(array('A','B','C'));
select explode(array('A','B','C')) as col;
select tf.* from (select 0) t lateral view explode(array('A','B','C')) tf;
select tf.* from (select 0) t lateral view explode(array('A','B','C')) tf as col;
```

col
A
B
C

explode (map)

```
select explode(map('A',10,'B',20,'C',30));
select explode(map('A',10,'B',20,'C',30)) as (key,value);
select tf.* from (select 0) t lateral view explode(map('A',10,'B',20,'C',30)) tf;
select tf.* from (select 0) t lateral view explode(map('A',10,'B',20,'C',30)) tf as
key,value;
```

key	value
A	10
B	20
C	30

posexplode (array)

```
select posexplode(array('A','B','C'));
select posexplode(array('A','B','C')) as (pos,val);
select tf.* from (select 0) t lateral view posexplode(array('A','B','C')) tf;
select tf.* from (select 0) t lateral view posexplode(array('A','B','C')) tf as
pos,val;
```

pos	val
0	A
1	B
2	C

inline (array of structs)

```
select inline(array(struct('A',10,date '2015-01-01'),struct('B',20,date
'2016-02-02')));
select inline(array(struct('A',10,date '2015-01-01'),struct('B',20,date
'2016-02-02'))) as (col1,col2,col3);
select tf.* from (select 0) t lateral view inline(array(struct('A',10,date
'2015-01-01'),struct('B',20,date '2016-02-02'))) tf;
select tf.* from (select 0) t lateral view inline(array(struct('A',10,date
'2015-01-01'),struct('B',20,date '2016-02-02'))) tf as col1,col2,col3;
```

col1	col2	col3
A	10	2015-01-01
B	20	2016-02-02

stack (values)

```
select stack(2,'A',10,date '2015-01-01','B',20,date '2016-01-01');
select stack(2,'A',10,date '2015-01-01','B',20,date '2016-01-01') as (col0,col1,col2);
select tf.* from (select 0) t lateral view stack(2,'A',10,date
'2015-01-01','B',20,date '2016-01-01') tf;
select tf.* from (select 0) t lateral view stack(2,'A',10,date
'2015-01-01','B',20,date '2016-01-01') tf as col0,col1,col2;
```

col0	col1	col2
A	10	2015-01-01
B	20	2016-01-01

Using the syntax "SELECT udtf(col) AS colAlias..." has a few limitations:

- No other expressions are allowed in SELECT
  - SELECT pageid, explode(adid_list) AS myCol... is not supported
- UDTF's can't be nested
  - SELECT explode(explode(adid_list)) AS myCol... is not supported
- GROUP BY / CLUSTER BY / DISTRIBUTE BY / SORT BY is not supported
  - SELECT explode(adid_list) AS myCol ... GROUP BY myCol is not supported

Please see LanguageManual LateralView for an alternative syntax that does not have these limitations.

Also see Writing UDTFs if you want to create a custom UDTF.

explode

`explode()` takes in an array (or a map) as an input and outputs the elements of the array (map) as separate rows. UDTFs can be used in the SELECT expression list and as a part of LATERAL VIEW.

As an example of using `explode()` in the SELECT expression list, consider a table named myTable that has a single column (myCol) and two rows:

Array<int> myCol
[100,200,300]
[400,500,600]

Then running the query:

```
SELECT explode(myCol) AS myNewCol FROM myTable;
```

will produce:

(int) myNewCol
100

200
300
400
500
600

The usage with Maps is similar:

```
SELECT explode(myMap) AS (myMapKey, myMapValue) FROM myMapTable;
```

posexplode

> **Version**
> Available as of Hive 0.13.0. See HIVE-4943.

`posexplode()` is similar to `explode` but instead of just returning the elements of the array it returns the element as well as its position in the original array.

As an example of using `posexplode()` in the SELECT expression list, consider a table named myTable that has a single column (myCol) and two rows:

Array<int> myCol
[100,200,300]
[400,500,600]

Then running the query:

```
SELECT posexplode(myCol) AS pos, myNewCol FROM myTable;
```

will produce:

(int) pos	(int) myNewCol
1	100
2	200
3	300
1	400
2	500
3	600

json_tuple

A new json_tuple() UDTF is introduced in Hive 0.7. It takes a set of names (keys) and a JSON string, and returns a tuple of values using one function. This is much more efficient than calling GET_JSON_OBJECT to retrieve more than one key from a single JSON string. In any case where a single JSON string would be parsed more than once, your query will be more efficient if you parse it once, which is what JSON_TUPLE is for. As JSON_TUPLE is a UDTF, you will need to use the LATERAL VIEW syntax in order to achieve the same goal.

For example,

```
select a.timestamp, get_json_object(a.appevents, '$.eventid'),
get_json_object(a.appenvets, '$.eventname') from log a;
```

should be changed to:

```
select a.timestamp, b.*
from log a lateral view json_tuple(a.appevent, 'eventid', 'eventname') b as f1, f2;
```

parse_url_tuple

The parse_url_tuple() UDTF is similar to parse_url(), but can extract multiple parts of a given URL, returning the data in a tuple. Values for a particular key in QUERY can be extracted by appending a colon and the key to the partToExtract argument, for example, parse_url_tuple('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUERY:k1', 'QUERY:k2') returns a tuple with values of 'v1','v2'. This is more efficient than calling parse_url() multiple times. All the input parameters and output column types are string.

```
SELECT b.*
FROM src LATERAL VIEW parse_url_tuple(fullurl, 'HOST', 'PATH', 'QUERY', 'QUERY:id') b
as host, path, query, query_id LIMIT 1;
```

GROUPing and SORTing on f(column)

A typical OLAP pattern is that you have a timestamp column and you want to group by daily or other less granular date windows than by second. So you might want to select concat(year(dt),month(dt)) and then group on that concat(). But if you attempt to GROUP BY or SORT BY a column on which you've applied a function and alias, like this:

```
select f(col) as fc, count(*) from table_name group by fc;
```

you will get an error:

```
FAILED: Error in semantic analysis: line 1:69 Invalid Table Alias or Column Reference
fc
```

because you are not able to GROUP BY or SORT BY a column alias on which a function has been applied. There are two workarounds. First, you can reformulate this query with subqueries, which is somewhat complicated:

```
select sq.fc,col1,col2,...,colN,count(*) from
  (select f(col) as fc,col1,col2,...,colN from table_name) sq
 group by sq.fc,col1,col2,...,colN;
```

Or you can make sure not to use a column alias, which is simpler:

```
select f(col) as fc, count(*) from table_name group by f(col);
```

Contact Tim Ellis (tellis) at RiotGames dot com if you would like to discuss this in further detail.

UDF internals

The context of a UDF's evaluate method is one row at a time. A simple invocation of a UDF like

```
SELECT length(string_col) FROM table_name;
```

would evaluate the length of each of the string_col's values in the map portion of the job. The side effect of the UDF being evaluated on the map-side is that you can't control the order of rows which get sent to the mapper. It is the same order in which the file split sent to the mapper gets deserialized. Any reduce side operation (such as SORT BY, ORDER BY, regular JOIN, etc.) would apply to the UDFs output as if it is just another column of the table. This is fine since the context of the UDF's evaluate method is meant to be one row at a time.

If you would like to control which rows get sent to the same UDF (and possibly in what order), you will have the urge to make the UDF evaluate during the reduce phase. This is achievable by making use of DISTRIBUTE BY, DISTRIBUTE BY + SORT BY, CLUSTER BY. An example query would be:

```
SELECT reducer_udf(my_col, distribute_col, sort_col) FROM
(SELECT my_col, distribute_col, sort_col FROM table_name DISTRIBUTE BY distribute_col
SORT BY distribute_col, sort_col) t
```

However, one could argue that the very premise of your requirement to control the set of rows sent to the same UDF is to do aggregation in that UDF. In such a case, using a User Defined Aggregate Function (UDAF) is a better choice. You can read more about writing a UDAF here. Alternatively, you can user a custom reduce script to accomplish the same using Hive's Transform functionality. Both of these options would do aggregations on the reduce side.

Creating Custom UDFs

For information about how to create a custom UDF, see Hive Plugins and Create Function.

select explode(array('A','B','C'));select explode(array('A','B','C')) as col;select tf.* from (select 0) t lateral view explode(array('A','B','C')) tf;select tf.* from (select 0) t lateral view explode(array('A','B','C')) tf as col;